Aalto University

School of Science

Master's programme in Computer, Communication and Information Sciences

Joni Makkonen

Performance and usage comparison between REST and SOAP web services.

Master's Thesis

Espoo 12.11.2017

Supervisors: Antti Ylä-Jääski

Aalto University
School of Science

Abstract of master's thesis

AALTO UNIVERSITY
SCHOOL OF SCIENCE

Author Joni Makkonen		
Title of thesis Performance and usage comparison between REST and SOAP web services.		
Degree programme Master's programme in Computer, Communication and Information Sciences		
Thesis supervisor Antti Ylä-Jääski	**Code of professorship** SCI3042	
Major Computer Science		
Thesis advisor(s)		
Date 12.11.2017	**Number of pages** 7+44	**Language** English

REST and SOAP are web service technologies for solving the message delivery problem. The choice between the two is not clear and comparison is difficult. This thesis tries to do the comparison and ease the choice with the recommendations. Also the aim of this work is to research REST as a replacement for SOAP for Seitatech Payment solution.

The definitions of SOAP and REST and the usage of both is described. The definition studies are used to do the comparison in a conceptual and feature level. In addition, practical tests about the performance of each technologies is made. A simple test setup is created using Seitatech provided web service platform. Afterwards, the test results are analysed.

The test results show REST to outperform SOAP in terms of bandwidth usage and message processing performance. During the test cases, performance issues was discovered when message size grows, which indicates parser issues in Seitatech platform.

The comparison provided results of characteristic differences between SOAP and REST. The recommendation of REST is made in most common cases, as it is less complex, less burdening and easier to develop and use than SOAP. SOAP should only be chosen if particular functionality, such as security options, is required.

Keywords REST, SOAP, comparison, web service, performance, usage

Aalto-yliopisto
Perustieteiden
korkeakoulu

Diplomityön tiivistelmä

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

Tekijä Joni Makkonen		
Työn nimi: REST ja SOAP pohjaisien web-palveluiden käyttö ja suorituskyky		
Koulutusohjelma Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma		
Valvoja Antti Ylä-Jääski	**Koodi** SCI3042	
Pääaine Tietotekniikka		
Ohjaajat		
Päivämäärä 12.11.2017	**Sivumäärä** 7+44	**Kieli** Englanti

Web-palveluiden kehityksessä viestien kuljetus järjestelmässä on merkittävä ongelma. REST ja SOAP ovat teknologioita, mitkä vastaavat tähän ongelmaan. Valinta näiden teknologioiden kesken on vaikea, sillä REST ja SOAP ovat tyyliltään erilaisia ja haastavia verrata keskenään. Tämä työ pyrkii tekemään vertailun näiden teknologioiden kesken ja helpottamaan tätä valintaa. Tämän työn tarkoitus on myös tutkia REST pohjaisen web-palvelun potentiaalia korvaamaan SOAP pohjaista palvelua.

Tässä työssä käydään läpi REST ja SOAP teknologioiden määritelmät. Näiden määritelmien avulla vertaillaan järjestelmiä keskenään sekä määritelmä, että toiminnallisuus tasolla. Määritelmävertailun lisäksi suoritetaan käytännön testejä, millä pyritään löytämään mahdolliset suorituskykyerot. Näitä testejä varten Seitatech on tarjonnut alustan, mitä muokkaamalla testit saadaan suoritettua.

Käytännöntestit osoittavat REST arkkitehtuurin suoriutuvan paremmin sekä viestien prosessoinnissa että kaistankäytössä. Testien aikana saatiin myös tietoa Seitatechin alustasta, missä huomattiin ongelmia viestien käsittelyssä kun viestien koko kasvoi suureksi.

Vertailun lopputuloksena osoitettiin REST pohjaisen järjestelmän sopeutuvan paremmin yleisimmissä tilanteissa. Suorituskyvyn lisäksi REST määritellään yksinkertaisemmaksi ja helpommaksi kehittää ja käyttää, kun taas SOAP on yleisesti rajoitetumpi ja raskaampi viestien siirtoon. SOAP kuitenkin tarjoaa laajemmat työkalut ja laajennukset, jolloin se voi olla soveltuvampi ratkaisu esimerkiksi turvallisuutta ja luotettavuutta vaativissa järjestelmissä.

Avainsanat REST, SOAP, vertailu, web-palvelut, suorituskyky

Table of contents

Preface

I thank the Professor Antti Ylä-Jääski for assisting with this thesis. I would also like to thank Seitatech personnel for help and providing the required testing platform and ideas for test cases.

Abbreviations

API	Application Programming Interface
FTP	File Transfer Protocol
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
URL	Uniform Resource Locator
XML	eXtensible Markup Language

1. Introduction

Web developers today have a burden of choosing the correct technologies for their solutions. Not everything is done from the scratch, so even more complete options are used, thus providing ease of creation of new software. At the same time the available software to choose from is growing. Similarly to software, the design issues must be taken into account, requiring additional level of decisions.

When creating web services, it is important to describe communications in a structural way. The technologies used must be chosen and there are multiple choices to choose from. Two popular options are SOAP and REST, which answer the communication question by providing guidelines for creating messages and communication channels between the components of the web service.

REST and SOAP have been targets for comparison and discussion for some time. Instead of SOAP, many of the web services today use REST as the architecture approach, which was originally proposed by Roy Fielding in his PhD dissertation. There are various differences between the approaches and applying new is always risky. [1, 18, 23]

The comparison is difficult, as the SOAP and REST are not equally the same. REST is more of a guideline for developers to give directions how to plan their web services, while SOAP is standardized protocol. These two are still seen on a table when planning new API for web service, so the need for comparison is real. This thesis tries to do the comparison and ease the decision making between SOAP and REST.

Objective of the work and research questions

The purpose of this work is to review RESTful architecture and SOAP protocol, and compare them when used to deploy web services. In addition, this work aims to find out when each of the technologies should be used. Especially the point of view of payment web service is taken in to account, where the Seitatech payment application is used as a testing platform.

In this study, we attempt to answer to the following questions:

- What are the advantages and disadvantages of RESTful architecture? These questions are aimed for payment web service used by Seitatech.
- The REST in an architecture, while SOAP is a message delivery protocol. How the comparison is possible?
- Which of the technologies should be chosen when developing new web service?

Seitatech has provided an SOAP interface, which is planned for rework. In this thesis, I am using their provided web service platform to run practical tests for metrics to support the comparison. The results from the tests are used together with similar studies to get information of the performance between the technologies. Also, the results are used to give recommendations for Seitatech for planned rework.

The technologies are analyzed in definition level to display weaknesses and strengths detailed in specifications and creating an estimate based to only theoretical aspects. The results of these analysis are targeted for the use of future projects and to enhance overall knowledge.

Structure of the work

The rest of the thesis is organized as follows. Chapter 2 provides the definition of the REST and a way to apply it to a web service. Chapter 3 reviews the second main technology in the work: SOAP. In chapter 4 we take a quick look to the Seitatech payment web service and the tests the interface is used for. Before the conclusion, chapter 5 concentrates to compare the two technologies using their definitions while also applying the payment application point of view. Finally, the conclusions summaries the main points of the work and gathers the vital key findings in this thesis.

2. Representational State Transfer (REST)

In this chapter we dive into the theoretical part of this study and review the definition of Representational State Transfer (REST). We use the work of the Roy Thomas Fielding to get the clear and compact explanation of the main points of the REST architecture. The second part of the chapter introduces the concept of Resource Oriented Architecture (ROA), which is Leonard Richardsons and Sam Rubys way to bring the potency of REST to practice. [1, 2, 31]

REST is a software architecture style bringing guidelines for creation of resource based web services. As a commonly seen, web services require support for heavy load balancing, where REST is able to answer the need with scalability as its strength. Also, the REST concentrates on commonly used technologies, such as HTTP and JSON, which brings simplicity to the software design and makes the use of the new architecture easier. In performance point of view the REST is capable of giving more than more of the heavy weight competitors. [32]

2.1 Definition of REST

Fielding defines REST as *"architectural style for distributed hypermedia systems"* and the whole definition consists of constraints and the architectural elements. Constraints is a set of characteristics or rules the web service is fulfilling to bring up the strengths of REST. In addition to constraints, the REST distinguishes three architectural element classes: Data elements, connectors and components. Data elements define the structure of the information transferred through the REST interface. This information receives definition how it should be presented in RESTful web service. Connectors and components are the construction components of the services. Architectural views are used to show the system as a whole using the previously mentioned elements. [1]

2.1.1 Constraints

For REST architecture there are constraints defined to bring in the characteristics: Client-Server, Stateless, Uniform interface, Cacheable, Layered system and Code on demand. These constraints are meant to be the guidelines, which when followed, bring the advantages like scalability and simplicity to your service. [29, 37, 40]

The first constraint, Client-Server, appoints the separation of user interface from server systems. The main conceit of the constraint is to allow the designing of client and server parts individually, thus forming portable interfaces and improving the scaling of server resources. Also, the Client-Server constraint allows separate updating of the components, which creates flexibility to development and upkeep of the software.

Stateless –constraint is related to the Client-Server style by forcing each request to contain all of the necessary information to make it processable. The requests may not reference to any of the previously run context. In practice, this means that the state is only kept with the client side

allowing server services to focus on single task at time. The constraint eases the designing and development of the software by allowing server side functionality to be implemented simultaneously. In addition, the constraint brings reliability and visibility to the request handling, causing less error situations and makes them easier to solve while also making the client side easier to develop. On the other hand, the constraint includes disadvantage of increasing network traffic due the increased need for messages. [44]

Cacheable is the third constraint of the REST, which on the contrary of the Stateless -constraint bring efficiency to network traffic by forcing responses to be tagged as cacheable or non-cacheable. While the server must add the cacheable tag to response, it allows client to use the response in future with the similar requests, thus reducing the amount requests sent to server. The disadvantage in caching is the decrease of reliability as the use of deprecated data enables possibility for unwanted situation. In practice, caching causes additional effort requirement to software design, causing increased complexity and extra difficulty for development.

The next constraint, Uniform interface, is most likely the one of the most important characteristic of the REST for the developer. The aim of the uniform interface is to simplify and bring visibility to the interactions with the small disadvantage of efficiency caused by standardization. The benefit of this constraint applies to developer as well as the user of the web service, making it highly valued in web services. The downside in this is the minor efficiency degradation, as the information is no longer sent in program specific format.

Table 2.1 Example of uniform and non-uniform interface functions.

Uniform interface	**Non-uniform interface**
Get(URI)	getCustomer()
Put(URI, Resource)	updateCustomer()
Delete(URI)	deleteCustomer(id)

The REST defines four aspects for interface to gain the advantage of uniform interface:

- identification of resources
- manipulation of resources through representation
- self-descriptive messages
- hypermedia as the engine of application state

In Table 2.1 is presented an example of similar interface with possible uniform and non-uniform functions. In the example developer and the user can easily identify the action currently being done, while the target of the action is added as parameter and uses the readability of URIs to keep the interface readable. [41]

The next constraint is layered system. The layered system style separates the components even more from each other by preventing them to see behind the other components. The components interact with the components in the nearby layers and only with them. This creates possibility to encapsulate the functionality of the system and to hide them from the client side. This provides extra security and simplicity for software design, while also enabling an easy way to add new functionality. The downside with extra layers comes with the overhead and latency of processing.

The final constraint of REST is style of Code-on-demand. This means that REST software is allowed to extend with the use of external code. This reduces the implementation time for client side as the commonly used code may be downloaded and used. The disadvantages with the constraint is the reduced visibility. The code-on-demand is the only optional constraint defined for REST.

The constraints define the base of the REST. Each constraint can be applied separately, but to create true RESTful application all of them must be used. The base idea of REST is to provide integrity through these constraints and applying only subset of them may cause the loss of this feature. The summary of the architectural design of REST can be found in table 2.2.

Table 2.2. Constraints of REST with short explanations.

Constraint	**Explanation**
Client-Server	Client interface is separated from server
Stateless	Request contains all information to make it processable
Cacheable	Responses may be used again in client side
Uniform interface	Interface is defined using simple HTTP methods and URIs
Layered system	System components are separated and layered
Code on demand	Use of external code is allowed

2.1.2 Data elements

Data elements are the second key aspect of REST architecture and first of the architectural elements. The communication between the components is created through the representations of the resources. These representations are formed of the set of standardized data types that are selected by the required functionality of service. The representations hide the details of data type to provide a common interface for the resource transfer. The REST defines six data elements, which are presented in table 2.3. [1, 37, 39]

Table 2.3. REST data elements.

Data Element	Description	Example
Resource	Anything that can be addressed with URI	A document, image
Resource identifier	A name which identifies the resource, URI	domain.example.com/document
Representation	An entity which is sent between clients and server.	HTML document
Representation metadata	Description of representation	Media type
Resource metadata	Description of resource. Provides information like location or additional source identifiers.	Source link
Control data	Defines the purpose of the message.	If-Modified-Since, If-Match

To completely understand the REST as an architecture, the concept of a resource must be defined. In REST, the resource is any kind of information or concept that can be named: a document, an image, service, person, a set of any of these resources and so on. The only requirement for the resource is that you must be able to target it and separate from other resources. The resources can be dynamic or static type. The nature of dynamic type resource is changing, meaning that accessing the dynamic resource may provide different resource in different time of access. On the contrary, the static resource will always return the same resource. For example, the “latest version” of a document is changed after a new version is released, thus being dynamic type of resource. By referencing resource with “doc_version_0.9”, the document remains unchanged as the new version is being named as “version_X.X” and we are accessing it static way.

The ability to hide resource information behind the name of the resource enables key features of web architecture. First, the generality is reached by the information of identifiers, which allow identifying of resource without revealing the actual type of resource. Secondly, the feature allows content to be bound to representation after reading the requirements of the request. Third advantage comes with freedom of referencing the resource, which, when identifiers are correctly used, removes the need to update the links when representation changes.

In RESTful resource access, the unique identifier tells the type of the resource the client requests. With web-based systems, the identifier is usually represented using Uniform Resource Locator (URL). The naming of the identifier should be done to describe the nature and position

of the resource, thus creating visibility and understandability for the resource requested. The following could be resources in web store –type service:

- www.example-store.com/customers
- www.example-store.com/12345/orders
- www.example-store.com/item/54321

The state of the resource is captured using the representations. The representation in REST is form of way to describe the data of the resource and use it to transfer resources between the components. The representation consists of the resource data, metadata and possibly some form of message integrity data. JSON or XML in case of web services are commonly used examples of representations for RESTful services.

Control data serves the way to give parameters to requests or responses and is used to perform actions to messaging. For example, a request could include control data to disable caching behavior.

2.1.3 Connectors

The second architectural element of the REST is connectors, which present the interfaces between the components. The connectors manage the communication and resources for the components, bringing abstraction by hiding the complex implementation. Roy Thomas Fielding presents following types of the connectors: client, server, cache, resolver and tunnel. [1]

Through the abstraction, the connectors bring simplicity and effectiveness to the system. The connectors allow component to exchange information between the requests increasing the efficiency and responsiveness. As the connector hides implementation of communication management, an individual system can be replaced without affecting the use of it.

The basic connectors are the server and client. These connectors perform like normal server-client models: client sends requests and server listens for them. The commonly used *Libwww* -library is an example for server and client connectors.

Cache is the next connector type, which can be used among the client or server connectors. As the cache -connectors name states, the connector allows storing the responses for future use, thus bringing the effectiveness to request handling through the reduced amount of sent requests. Cache is usable in both client and server side to improve the latency for either. Client caching is extremely beneficial as the response is received immediately without applying any network traffic. Server caching can also bring latency improvements in internal networks, while also providing less performance peaks on requests. For example, browser cache can be described as client cache.

The fourth connector type, the resolver, translates resource identifier to network address to create the connection to the requested resource. This can be implemented as hostname translation to get the IP address of the server, or as resource identifier translation to get changing resource using more static resource identifier.

2.1.4 Components

REST divides the components in four classes: origin server, user agent, proxy and gateway. The components link to their proper connectors to communicate with each other.

The first component is origin server, which acts as a source of the resources in its namespace. Using the server connector the origin server provides an interface for the resource accessing and receives the requests through it for resource modifications.

The client-side component, user agent, serves as request initiator and receiver of response. As the origin server uses server connector for its communication, similarly the user agent is tied to the client controller.

The rest of the components, a proxy and gateway, forward the initiated requests and responses, thus serving the end components both as a client and server side. A proxy is client-side packet translator, providing data translation, performance and security. Proxy can also be used to encapsulate other services for simplifying the client-side interface. A gateway provides similar functionality as proxy, but serves in the server side. The difference between these components lies in the way they are used: client side will decide when proxy is used, while server side always uses gateway when it is present.

2.1.5 Architectural views

Roy Thomas Fielding [1] also describes architectural views to illustrate the REST as an architecture. Three architectural views are described: process view, connector view and data view.

The first view, process view, displays the data flow between the client and server components. In figure 2.1 we can see a simple example of what in the process view can be included. The example shows a client connected to server through multiple routes: direct access to server and access through proxy and gateway. Here the server and client components are separated and system layered, allowing adding of intermediaries affecting the system interfaces. The intermediaries may be used for many reasons: data translation, performance improvement, service encapsulation and so on. Note that all the data paths are independent and have no interaction with each other, which is possible due the stateless nature of REST.

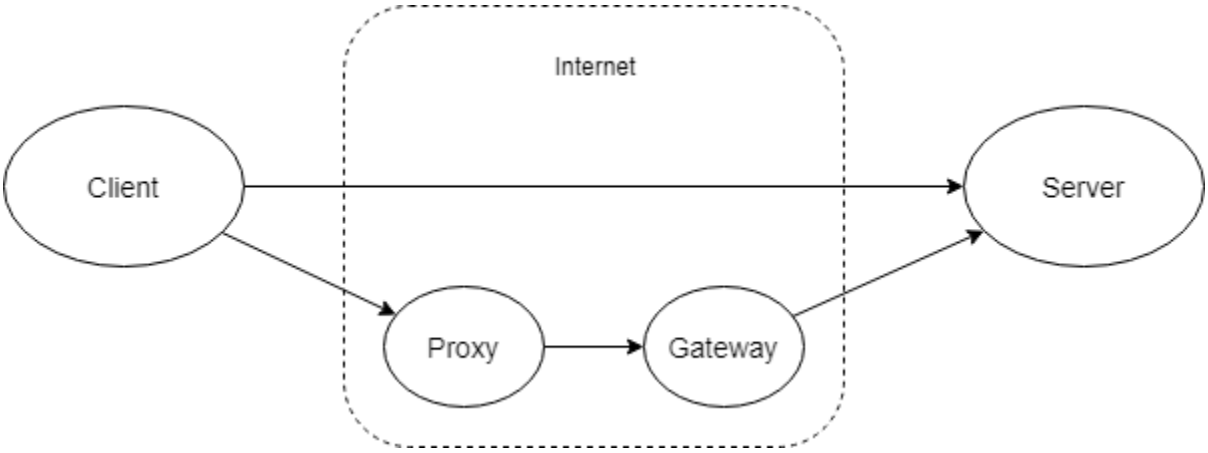

Figure 2.1. Process view of simple client-server architecture.

The communication between the components are described in connector views. The connector view is close to interface view, where the protocols used are specified. The REST allows freedom of choice to communication protocols, but still the interface between the components is limited. For example, in picture 2.2 the communications are processed using HTTP, but additional protocols FTP and STMP are also included to architecture.

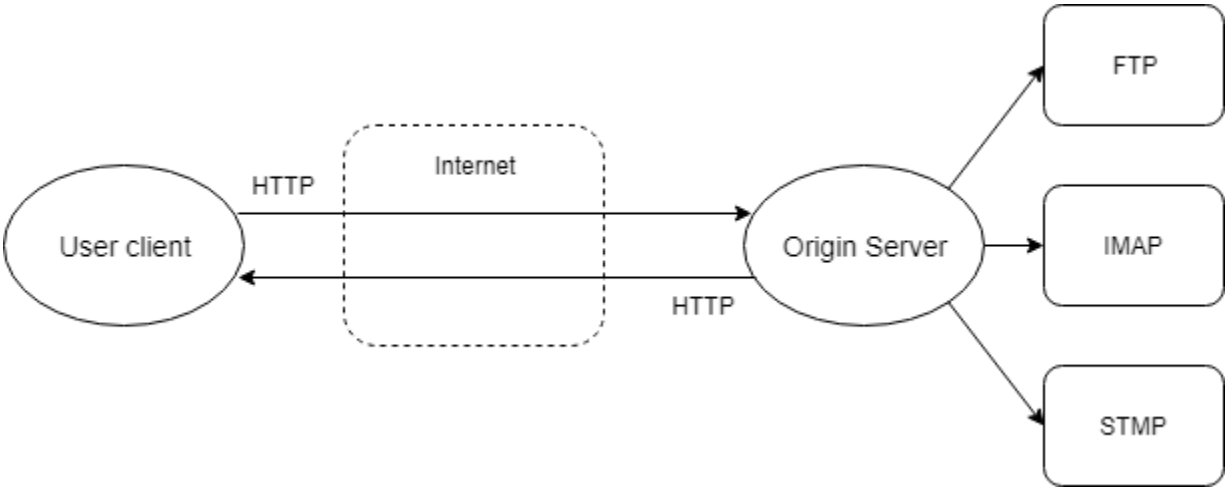

Figure 2.2. REST architecture with multiple protocols used.

The last architectural view for REST displays the architecture as an application. The data view shows the application as a structure of information and a system processing certain task with some user input.

2.2 Applying REST to Web Services

The REST provides the guideline for designing a web service, but still requires more concrete rules to actually build one. The Resource-Oriented Architecture (ROA) is an architecture, which applies all the strengths of the REST and provides a way to turn the guideline into an architecture. In their study, Richardson and Ruby define ROA as follows: “The ROA is a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web, and that programmers will enjoy using”. [2, 21, 22, 42]

The ROAs effective parts are the resources and more specifically, they are the names, representation and the links between them. There are two main features of ROA: addressability and statelessness, which along with URIs, HTTP and XML works as a RESTful service. By capturing these strongpoints, we can apply them to web services’ ways to display and use the resources efficiently. [2]

2.2.1 Resources and URIs

As defined earlier with the definition of REST, resource is anything that can be referenced with a name. In ROA there is an additional requirement for resource to be a resource: it has to have at least one address that represents the resource. In ROA, these addresses are identified using URIs. If some resource in web does not have a URI, it is not accessible and does not actually exist [3]. The examples of the resources and their URIs could be:

- *www.example-service.com/logs/11-11-2015*
- *www.example-service.com/backup/2054*

The resources can have multiple URIs. For example, the following URIs can point to the same resources:

- *www.example-service.com/logs/11-07-2015*
- *www.example-service.com/logs/11072015*

Note that the URI can return the same data to client, but still point to different resources. For example, the fetching from following similar URIs

- *www.example-service.com/logs/11-7-2017*
- *www.example-service.com/logs/latest*

may provide same byte stream to client, but the URIs are still pointing to different resources.

Technically URIs are not required to have any structure or clarity, but to have a good web design a clear format is preferred. The main rule for URI is that it should be descriptive, which is achieved by building the structure of the resources in predictable way. This brings freedom to

users and allows them to use your service in a more different way. For example, to search for a product from a service provider we should not use URI like [4, 26, 28, 42]

- www.example.com/product/search/11526

but instead

- www.example.com/product/11526.

2.2.2 Addressability

Addressability is the first feature of the ROA. As REST constraint requires, addressability defines the application to expose all of its significant data as a resource. As the ROA presents resources using URI, every piece of data has at least one URI in the application.

For the end-users, addressable web services allows easier access to resource and even allows new ways to use them. Addressability allows user to bookmark resources explored through the web site and to return at them later. In addition, The URIs of the resources could be used as an input to the other systems, which would not be possible for non-addressable services.

2.2.3 Statelessness

The second feature of ROA is the statelessness. The definition of statelessness is similar to the REST constraint: the server of stateless application can handle the requests without need of information from previous requests. In other words, the requests sent by client include all the information to complete the request in server side.

Addressability states that every piece of significant data has an URI. By applying this to definition of statelessness, we can say that in stateless application every state in server is a resource and needs an URI of its own.

The statelessness adds stability and clarity to the web service as the server defines accepted site using resources. Also, the stability is improved as there is no need for stable connection between client and server. The server does not need to hold the state of the client and the client

Servers using multiple devices to even out the load benefits the statelessness property. As the requests are separated and the server does not need to worry about the state of the client, the request handling could be distributed freely between server hardware. Also the caching decisions are easier to implement in these systems due the possibility to check requests separately.

3. Simple Object Access Protocol (SOAP)

In this chapter we go through the definitions and overview of the competitor of REST: Simple Object Access Protocol (SOAP). After brief overview, we go through the definition of SOAP as it is specified in W3C standardization. We also take advice from James Snell by following his understanding of SOAP and how it is applied to create web services. [5, 6, 16, 34, 35]

SOAP is our second answer to designing API for web service, where the SOAP defines platform free method to implement messaging [6]. The base of the SOAP relies on XML message format and HTTP for communications. The SOAP definition mainly describes a one-way message delivery, but it can be implemented to serve request-response communications, which is common way of usage of SOAP.

Web services using SOAP can be referred as RPC-style web services, where the message is sent in envelope and aims to carry out a task by running functions or methods. This RPC –styled SOAP uses messages to call functions or methods and uses the message to carry the function name and parameters, which after the return value is received. The alternative is to use SOAP to deliver messages as a document –style. This application is called an Electronic Document Interchange (EDI), which uses XML to hold documents like purchase orders or business transaction. In this part, we focus on explaining the RPC –style. [5, 43]

SOAP has two important versions used in production: 1.1 and 1.2. The differences between these versions are minor, but still the version mismatch causes problems with message processing. The main problem occurs when different versioned message is processed in another versions system. Version 1.2 can handle these situations better, as the system can decide whether to process version 1.1 message or generate fault message. Version 1.1 systems always generate fault when they receive version 1.2 messages. [5, 6]

3.1 SOAP Messages

3.1.1 Message structure

The messages in SOAP are formed by using the XML document format [7, 38, 43]. XML provides human and machine -readable message structure, which has widely been used as a document or data structure. The XML messages provide wide support for variety of applications, are easy to create and use and straightforwardly usable over the internet. An example 3.1 displays an example message of XML.

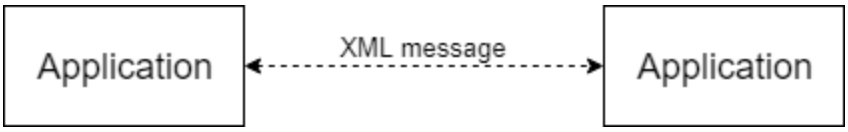

Figure 3.1. SOAP applications messaging through XML.

SOAP introduces messages with envelope element, which is the root element of the SOAP message. The envelope includes the core elements of the message: header and body. Every envelope must contain exactly one body element, while the header elements are optional. The body element may still contain as many sub elements as required. The structure of SOAP envelope is shown in figure 3.1.

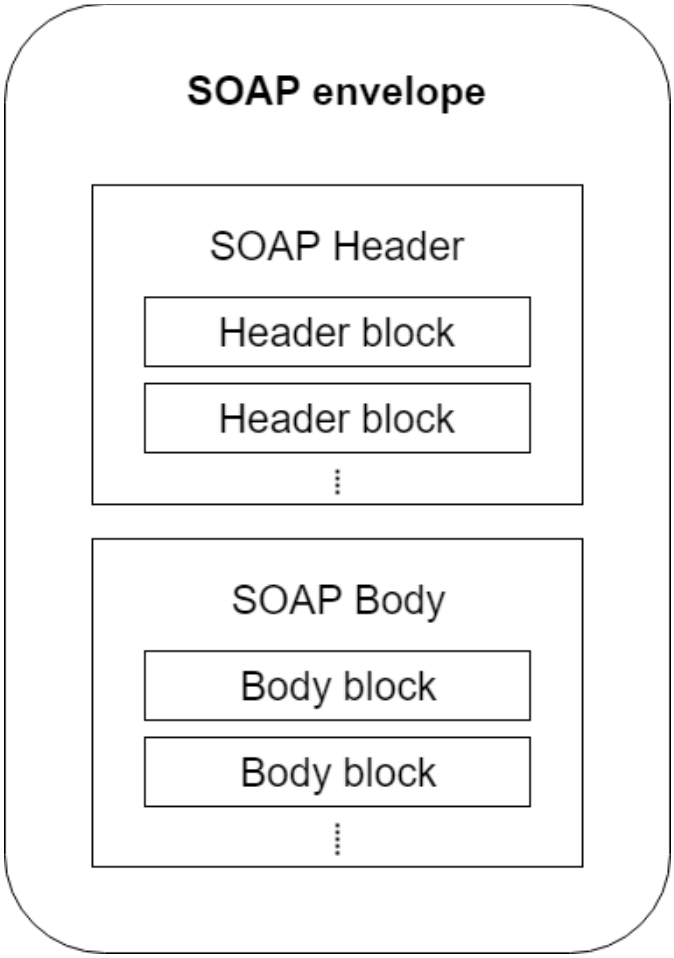

Figure 3.2. Structure of SOAP message.

The header is the optional section of the envelope and provides application extensibility and a way to add additional information. Including header allows application to create message processing logic to be targeted to specific nodes along the way to the endpoint. The data included in header usually contains information about the delivery and processing of the message, such as routing settings, processing instructions and authorization, but may also include message kind of data close to the actual message. The information added to header is application specific and depends on design and use of the application. For example, the header can include account number of a pay-per-use kind of service, where the intermediate node can use this header data to confirm the access to the service requested.

The body is the actual data of the message, which is targeted to the ultimate receiver. The body is mandatory and it may contain zero to any number of elements with the format specific to the application. The content of the body is application specific and may be defined as the required by the user. Simple example of SOAP message can be seen in example 3.1.

3.1.2 Message exchange

The most typical way to apply SOAP is the basic request-response message exchange between two nodes. The conversation between these nodes can be modelled as a document style or as a Remote Procedure Calls (RPC) as mention before. The choice between the styles is decided by the needs of the application and design. The RPC style provides more programmatic behavior and is used to directly invoke method or procedure calls, while the document style is more common when there is a need for richer data types.

The syntax of SOAP message is based on the http://www.w3.org/2001/06/soap-envelope namespace and is required to be included in envelope of SOAP message. This structure can be seen in examples 3.1 and 3.2.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <c:customer xmlns:c="http://www.examplestore.net/customers">
        <c:customerId> 1562754 </c:customerId>
        <c:customerName> Example Customer </c:customerName>
  </c:customer>
 </env:Header>
 <env:Body>
        <s:purchase xmlns:s="http://www.examplestore.net/store">
              <s:productName> Example product </s:productName>
              <s:productId> 12345 </s:productId>
        </s:purchase>
```

```
 </env:Body>
</env:Envelope>
```
Example 3.1. Document style request in web store

The example 3.1 presents the structure of the document -style SOAP message as a purchase request in an online store. As explained earlier, the message starts with envelope with namespace env="http://www.w3.org/2003/05/soap-envelope" introduced in it, which is requirement for SOAP version 1.2 and must be introduced or the SOAP nodes do not understand the message. The header of the example includes a namespace and the data of the customer similar to account data to most of the web services. In addition, the header introduces the role and mustUnderstand values, which are explained later in processing chapter. The body as well includes its own namespace and the payload for the ultimate receiver to process.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <c:customer xmlns:c="http://www.examplestore.net/customers">
        <c:customerId> 1562754 </c:customerId>
        <c:customerName> Example Customer </c:customerName>
  </c:customer>
 </env:Header>
 <env:Body>
        <s:purchase xmlns:s="http://www.examplestore.net/store">
              <s:productName> Example product </s:productName>
              <s:productId> 12345 </s:productId>
              <s:purchaseResult> Success </s:purchaseResult>
        </s:purchase>
 </env:Body>
</env:Envelope>
```
Example 3.2. SOAP response in web store.

The response for the online store request is presented in example 3.2. The response is similar to the request, except the added information to the body about the result of the purchase.

3.1.3 Remote Procedure Calls

SOAP applies the RPC procedure call protocol to enable complex structure for the message exchange. In client-server model, the client is the caller for the procedure and the server responds with the results.

SOAP 1.2 provides definition for packaging RPC in SOAP messages [6]. The RPC messages include the name of the invoked method and extra information given as parameters for it. To produce RPC message, the following information is needed:

- The address of the target SOAP node.
- The name of the procedure.
- All the input arguments required by the procedure or the output values in case of response message.
- Control information to make sure the correct web resource is reached.
- A clear pattern, which is used to transfer the message between the destinations. In practice, the HTTP method needs to be defined.
- Optional header data.

The structure of SOAP-RPC is presented in example 3.3. The example includes similar headers as in document-style messages, but includes *env:encodingStyle* attribute for message encoding specification. This attribute specifies the encoding scheme for the data structure, thus providing information that the contents are serialized according to the SOAP encoding rules. The body holds the actual RPC structure, which in this case invokes method *processPurchase* with product name and id as parameters.

```
<?xml version='1.0' ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Body>
       <s:processPurchase xmlns:s=http://www.examplestore.net/purchase
           env:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
              <p:product xmlns:p="http://www.mywebstore.net/products">
                     <p:productName> Example Product </s:productName>
                     <p:productId> 12345 </s:productId>
              </p:product>
              <c:customer xmlns:c="http://www.mywebstore.net/customers">
                     <c:customerId> 1562754 </c:customerId>
                     <c:customerName> Example Customer </c:customerName>
              </c:customer>
       </ s:processPurchase >
   </env:Body>
 </env:Envelope>
```

Example 3.3. SOAP-RPC request.

The RPC itself is always carried in body of the message. The process called is required to be delivered in structure defined by SOAP data model, where the first element includes the name of the procedure. The children of this element are the parameters. In example 3.3 the structure is displayed with an example purchase from online store, where the name of the process is

“processPurchase” with two parameters added under it: “product” and “customer”. Example 3.4 shows the corresponding response for the request. Note that the parameters are also data structures and may include any number of information fields. [8]

```
<?xml version='1.0' ?>
  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
  <t:transaction xmlns:c="http://www.examplestore.net/customers"
      env:encodingStyle="http:// www.examplestore.net /encoding"
      env:mustUnderstand="true">
  </t:transaction>
    </env:Header>
    <env:Body>
<s:processPurchaseResponse xmlns:s="http://www.examplestore.net/purchase">
          <p:resultCode xmlns:p="http://www.mywebstore.net/products">
                    1
          </p:resultCode>
</  s:processPurchaseResponse >
    </env:Body>
  </env:Envelope>
```

Example 3.4. SOAP-RPC response.

3.2 Nodes

SOAP processing model defines node as the main system component. The nodes are responsible for sending, receiving, forwarding and processing the messages sent in the system. Every node in SOAP based system has an URI, which is used to identify them.

Depending on the role and functionality of the node, the SOAP nodes are divided in the following types [6]:

- SOAP sender: Node that sends a message.
- SOAP receiver: Node that receives a messages
- SOAP intermediary: Node that receives the message and sends it. Also the intermediary may process the message between receiving and sending.
- Initial sender: Node that is the original source of the message.
- Ultimate receiver: Node that is final receiver of the message.

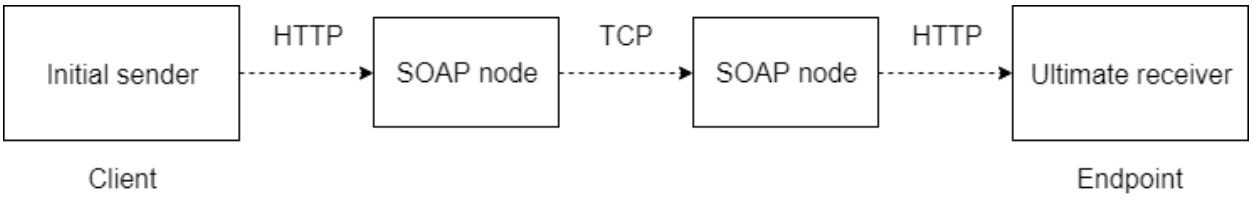

Figure 3.3. Example message pathing using changing transport protocols.

The SOAP intermediaries are separated to two types: forwarding intermediaries and active intermediaries. The forwarding intermediaries process the message according to the information in message header before sending the message to the next node. Active intermediaries perform the same procedure as forwarding intermediaries, but are also allowed to modify the received message in a way not described in the message header. These modification may have major impact on message handling in the following nodes.

3.3 Processing model

SOAP defines specific way for nodes to handle message processing. When node receives message, the header part is read and provides information about the logical processing of the message. When SOAP node receives message, the following actions are done to process the message [6]:

1. Confirm the syntax of the message to match SOAP message structure.
2. Process required header blocks determined by the role attributes.
3. If the node is intermediary, resend the message.
4. If the node is ultimate receiver, process the body of the message.

While intermediary processes headers, the SOAP defines rules on how the header blocks are forwarded. If intermediary processes header block, it is removed from the message, whereas the non-processed headers are left untouched. In the special case where the intermediary does not process the header block and the role attribute informs node to do so, the default behavior is to remove the header block. Note that even if the header block is removed, it may still be inserted back when processing the message.

SOAP defines three different attributes for the header blocks, which can be used to alter the default header block processing behavior: “role”, “mustUnderstand” and “relay”.

3.3.1 “role” attribute

The header blocks node needs to process may be included with env:role attribute, which tell the node if the header is required to be processed. When message is received and structure confirmed, the role block is read for each header and if the node assumes the role pointed by the env:role –attribute, the header block is processed. This allows skipping of headers and

targeting of header values certain nodes. The value of role can be customized by providing URI, which the node will identify. In addition, SOAP 1.2 defines three standardized roles for the use:

- "http://www.w3.org/2003/05/soap-envelope/role/none”
- "http://www.w3.org/2003/05/soap-envelope/role/next"
- "http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"

The processing requirements for the standardized roles can be found in table 3.1. For example, if intermediary node receives message with header using env:role –attribute set as “none”, the intermediary skips it and continues the processing.

Table 3.1. SOAP 1.2 standard roles and required actions by the nodes. [6]

Node/Role	Not present	“none”	“next”	“ultimateReceiver”
Initial sender	not applicable	not applicable	not applicable	not applicable
Intermediary	no	no	yes	no
Ultimate receiver	yes	no	yes	yes

The role attribute is not usable for body section, as the information of env:body is targeted for ultimate receiver, thus being skipped by intermediates.

3.3.2 “mustUnderstand” attribute

In addition to the env:role attribute, the header blocks can be included with env:mustUnderstand attribute, which applies additional processing for the nodes. If node receives header block with env:mustUnderstand, it must process the data inside or generate fault message. Env:mustUnderstand has a priority over other blocks, causing them to be processed before other blocks and in case of fault stopping the further processing of message. In example 3.5 the first header block includes both env:mustUnderstand and env:role attributes. As the env:mustUnderstand is set to “true”, the block must be processed, regardless of the env:role value.

```
<?xml version='1.0' ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <t:transaction xmlns:c="http://www.examplestore.net/customers"
        env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:mustUnderstand="true">
     </t:transaction>
   </env:Header>
   <env:Body>
```

```
        <s:processPurchaseResponse xmlns:s="http://www.examplestore.net/purchase">
            <p:resultCode xmlns:p="http://www.mywebstore.net/products">
                  1
            </p:resultCode>
        </ s:processPurchaseResponse >
    </env:Body>
  </env:Envelope>
```

Example 3.5.

3.3.3 "relay" attribute

The third of the SOAP 1.2 attributes is boolean -type of variable and indicates if the intermediate must send the header block forward if it is not processed. By default, the SOAP headers that are processed are removed from the message, so the "relay" -variable can be used to make sure the header block reaches every node along the way. In addition, by combining the "next" -role attribute with "relay", makes sure the header is processed in every node that receives the message.

3.4 Fault handling

The *env:body* has a separated attribute for handling error situations in SOAP messages: *env:Fault.* When any node fails to process message, it creates a "fault" block as an child of body. If the message already has "fault" block, new one should not be added. To be able to use this, the underlying transport protocol must be able to handle the response delivery.

The structure of faulty SOAP message is presented in example 3.6. As shown in example, the env:Fault has two mandatory elements: env:Code and env:Reason. In addition, the env:Code includes mandatory env:Value, where SOAP has five standardized error codes:

- "VersionMismatch": The SOAP envelope includes invalid namespace.
- "mustUnderstand": Node could not understand the message with "mustUnderstand" set as true.
- "Sender": Error with processing the message.
- "Receiver": Server error, where the contents of the message is not directly responsible of.
- "DataEncodingUnknown": The contents of the body or header block has unsupported encoding.

The application is also allowed to create custom elements under these codes to create more specific error information. For human readable error message, the env:Reason should be used. In addition, the fault may have env:Detail for more application specific information or env:Node to distinguish the creator node of the fault message. Also the role of the node may be added as env:Role element.

```
 <?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
        xmlns:rpc='http://www.w3.org/2003/05/soap-rpc'>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>rpc:BadArguments</env:Value>
        </env:Subcode>
      </env:Code>
        <env:Reason>
          <env:Text>Processing error</env:Text>
        </env:Reason>
        <env:Detail>
          <e:myFaultDetails
            xmlns:e="http://www.examplestore.net/faults">
            <e:message>Invalid product</e:message>
          <e:errorcode>999</e:errorcode>
        </e:myFaultDetails>
      </env:Detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Example 3.6. Faulty SOAP Message of example store. The message in detail indicates the wrong product id in the message.

3.5 Protocol binding

One of the features of SOAP includes the freedom to choose how the messages are delivered. The SOAP specifies the delivery of messages between the intermediate nodes as protocol binding. As the whole system is not bound to transport protocol, every node in the system can choose the transport protocol used to deliver the message. [6, 24]

Protocol binding defines details of the transport protocol used, but also the form of the data being delivered. This means that the SOAP messages in previous examples may be transformed from human readable XML to packed or encrypted form. For example, intermediaries that require extra security between the transfer, may encrypt the message transfer between them and relay the message normally afterwards.

The definition of protocol binding provides ways for additional features to SOAP application. The underlying protocol may not provide critical features required by the application, which still

may be implemented using SOAP headers. Using of headers also allows certain functionality to be forced as end-to-end feature, which ensure feature is provided even if underlying protocol does not provide in some point of message delivery.

Although the SOAP support any binding for message transport protocol, the SOAP 1.2 has standardized HTTP as the underlying protocol.

3.5.1 HTTP binding

W3C provided SOAP definition explains HTTP as an example binding for SOAP. HTTP is greatly suitable for request-response style communication, as it implicitly correlates the request message with response. [6]

The usage of HTTP binding is divided in two patterns: SOAP response message exchange and SOAP request-response message exchange. These two different patterns provide the World Wide Web styled message exchange functionality. The response message exchange relies on HTTP GET method to obtain the data of resource without altering it any way. The second pattern, request-response message exchange, applies HTTP POST to send and receive messages.

Similar to REST HTTP requirements, the SOAP 1.2 defines which of the two patterns should be used in certain situations. The HTTP GET –bound pattern is supposed to be in use when information is being retrieved only and no changes to database is applied, thus leaving database in same state as before it is before the request. This is referred as safe and idempotent method. The POST –method instead can be used in all situations, without restrictions. [8]

The example 3.7 presents the HTTP GET simplified request, where the resource "/www.examplestore.net/resource_name?param=1234" is requested. Note that the preferred response representation can be set with "Accept" –field, which in the example is set to *application/soap+xml*. The example 3.8 could be the response for the message, including both HTTP headers and SOAP response message, where the requested resource data is added to the body of the XML. Note that the RPC messages are not usable with GET –method, as the body and required information is missing from the request message.

```
GET /www.examplestore.net/resource_name?param=1234 HTTP/1.1
Host: www.examplestore.net
Accept: text/html;q=0.5, application/soap+xml
```

Example 3.7. HTTP GET request for SOAP response

When request-response pattern is used, the POST method is required. Similarly to GET – request, the POST adds the URI for the resource in first line of HTTP request. The SOAP specification requires this value to be valid URI, but set no other restriction to it. Note that the Content-Type –field must always have “application/soap+xml” –value when posting SOAP message. The example of POST –request can be seen in example 3.9. The response to POST request would be similar to the GET-request response seen in example 3.8 with the difference of envelope being filled with application specific response data instead of the resource.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=”utf-8”
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
   ...
   </env:Header>
    <env:Body>
     ...
    </env:Body>
</env:Envelope>
```

Example 3.8. Response to example 3.7 request.

In case of SOAP fault, instead of returning error code 200 “OK”, the 500 “Internal Server Error” should be returned. Also to the body of the XML response the “fault” block is added with appropriate information as required by SOAP. This error code can be misleading as the error happens in application level caused by client, but is still correct behavior by the definition. [9]

4. Test cases

In the first part of this chapter, we take a look to the interface provided by Seitatech, which is used as a target to estimate the advantages and disadvantages between SOAP and REST. The full functionality of the interface is not explained, but for simplicity we go through a simple transaction handling service as an example to provide a concrete example for the analyzes. Also the transaction handling is the most server burdening and the main task of the system, making analyzing of it more valuable.

In the second part, the test procedure and results we ran using the Seitatech provided platform are explained. The tests are simplified and aimed to enhance the performance of future Seitatech solutions and to provide performance test results of this implementation. These tests are the first part of the analytic part of this thesis, where second part is the theoretical comparison in chapter after this.

4.1 Background for Seitatech implementation

The architecture of the system is described in figure 4.1. The incoming data is received from payment terminals through the internet and received through the firewall to the first nodes. The Payment Gateways work as both forwarding and processing nodes, depending on message received. Also there is additional services where the messages may be forwarded. The processor is the final receiver of the transaction messages, which in this work handles authorizing the transactions and finally saving the completed transaction. The payment terminal acts as a client for the service, and the possible payment platform and Seitatech Payment Gateway act as an intermediaries before the message reaches the processor.

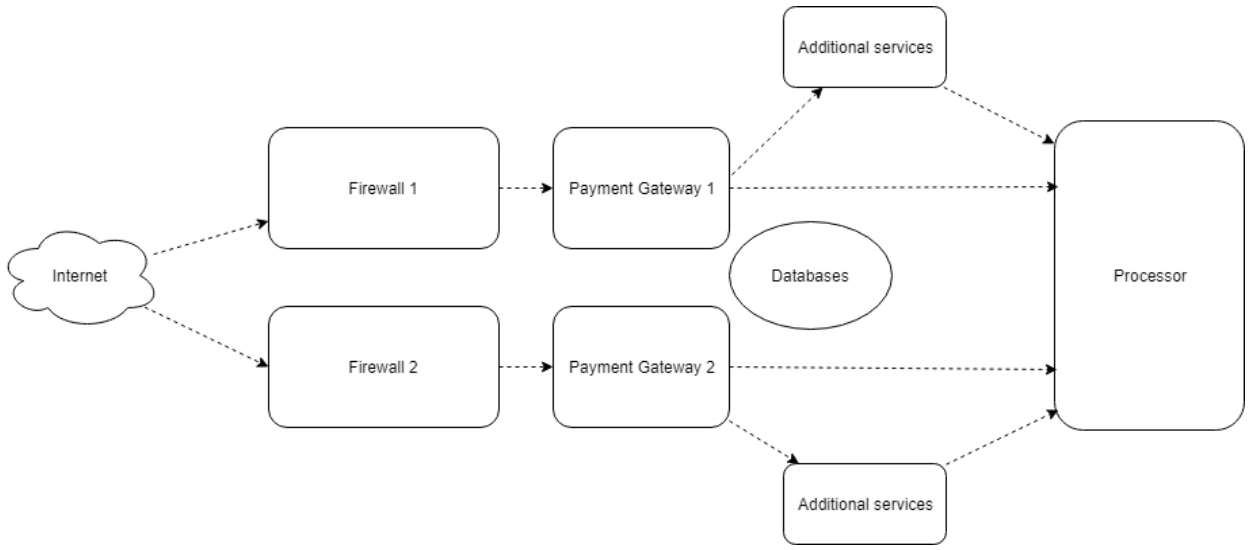

Figure 4.1. Architecture of Seitatech Payment Gateway.

The processing of transaction takes two messages: authorization and completion. Authorization is received when the terminal has obtained all the necessary information to do the transaction and requests now authorization from servers according the Finnish bank regulations. The authorization servers respond with message and depending the results terminal sends completion data. The completion data is checked and saved for later processing. This procedure can be seen in figure 4.2.

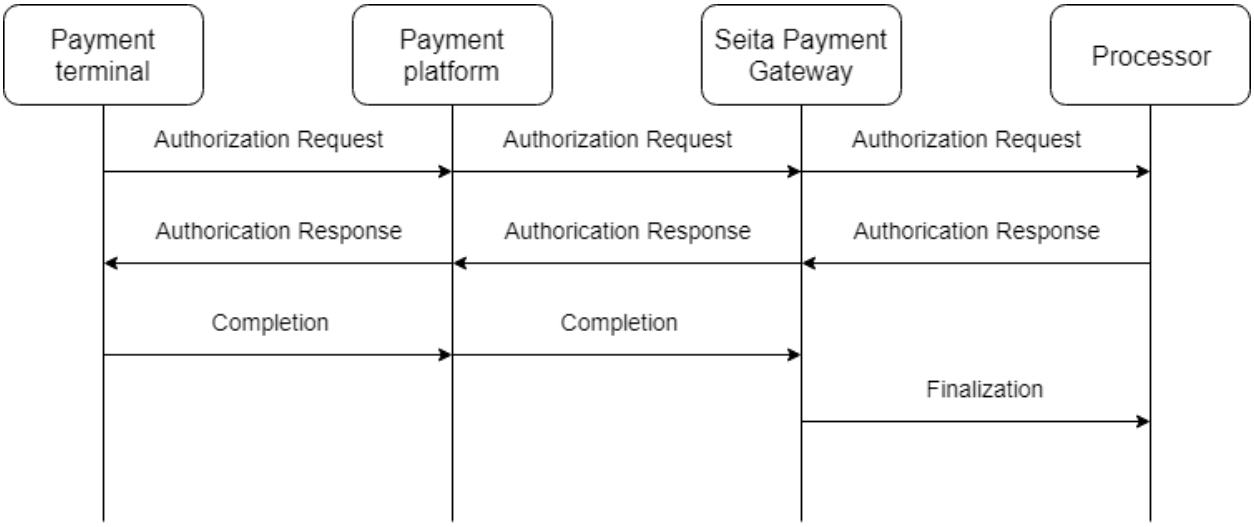

Figure 4.2 A successful transaction message flow.

As the interface handles payment information, the most important attribute is security. For this reason, all the messages are required to use SSL encryption. The second required property is the load handling, as the amount of incoming messages will grow during the increasing number

of payment terminals. To avoid over burdening the services, the Payment Gateway services are duplicated and the message flow distributed equally to both systems.

4.2 Test setup

Before the definition comparison, we do some practical testing to receive metrics of the performance between SOAP and REST. To get results relevant to the interface provided by Seitatech and also receive a straightforward comparison between SOAP and REST, we try to keep the tests as simple as possible with the platform in use.

When planning the testing, we identified metrics which were essential for the Seitatech. The chosen metrics we are interested in this practical testing are

- Bandwidth usage
- Processing time of the messages

Additional parameters, like CPU usage or memory usage, were considered, but to keep testing cases less complex we chose only the most important parameters. Also, the importance of the parameters were considered when doing this decision. The bandwidth was considered most important to test, as its capacity is more difficult to expand in practice as compared to adding additional hardware for extra CPU cycles or memory capacity. The processing time can still provide relative information about performance between the compared technologies, so it is less desired to know the actual usage of memory or processing performance.

To test these parameters in practice, we create a test web service and send simple messages to it. The web service is built on Seitatech web service platform, which applies Jetty -web server as a part of the web service solution. The web service supports both RESTful styled JSON messaging and SOAP XML messaging, so it is possible to run the tests in same environment for both technologies. The platform is created using Java and is fairly easy to modify and add the test cases, while allowing to run the web service application on various computer platforms. [17]

The tests are run by sending HTTP requests to the web service. For the metrics decided, two kind of requests are made for testing: fetching of data and sending of data. The first test is a light request with no heavy data included, which the web service is supposed to return a response with the requested heavier payload in appropriate form. For testing purpose, the request includes an amount of the data objects included in response, which allows us to request increasing amount of data. The second the test case includes sending a request with data similar of the first test case, which the server is supposed to parse and return a one line response for success. For clear results and to try to minimize randomness, all the tests are run

multiple times and the results averaged. The results are affected by several sources, like the current load of the web service machine or behavior of Java Virtual Machine.

The payload of the messages are random test data. The payload is created from objects including two lines of arbitrary test data of 42 bytes. The object created to web service are shown in example 4.1.

```
public class SoapTestData {

   private String Data1;
   private String Data2;

   public String getData1() {
      return Data1;
   }

   public void setData1(String Data1) {
      this.Data1 = Data1;
   }

   public String getData2() {
      return Data2;
   }

   public void setData2(String Data2) {
      this.Data2 = Data2;
   }
}
```

Example 4.1. Implementation of data object, which is used to increment size of the test messages.

The platform is set on separate machine running Windows 10, as the Seitatech test servers were not available for this work. The lightweight client application is run from the same machine, keeping the message travel times minimal.

To receive the metrics from client side, a custom application was created. This application was simple Windows Forms -application developed with C#, which allows use of different

parameters to allow flexible testing. For processing time, the application starts measuring time when creation of request is started and stops when response to the request is received. The message received is not parsed in client side. In addition, the application tracks the sent and received bytes to measure the network traffic between the endpoints.

The server side creates an endpoint for both SOAP and JSON messages to handle incoming requests. The endpoints receive the messages and parses them to the Java objects without processing them any further. For the testing, the Java classes are created to hold the payload in test messages. These classes are shown in examples 4.2 and 4.3. The SOAP parses the message in objects using WSDL file, which is generated during the startup of the service from Java classes. The JSON messages use google GSON to parse the received message to Java object and vice versa [28]. The GSON provides easy to use interface for parsing, requiring only calls of *fromJSON*- and *toJSON*- functions to switch the representation of the payload. The GSON is selected as it is already used around in Seitatech platform web services.

The test cases were chosen to run in following pattern to provide required test results:

- Both JSON and SOAP do two tests: request for payload and send the payload.
- Tests were run in two test data sizes: 50 to 500 data objects and 1000 to 19000 data objects. The size of the message alter from 10kb to 80kb and 150kb to 2,5 Mb.
- Each request is done 30 times and the average of their results is used.

The test sizes were chosen to match possible payment application messages, while also the large message give us extra information about the performance of the platform.

```
public class TestResponse {

   private String responseCode;
   private List<SoapTestData> responseData;

   public String getResponseCode() {
      return responseCode;
   }

   public void setResponseCode(String responseCode) {
      this.responseCode = responseCode;
   }

   public List<SoapTestData> getResponseData() {
```

```
        return responseData;
    }

    public void setResponseData(List<SoapTestData> responseData) {
        this.responseData = responseData;
    }
}
```

Example 4.2. Java class implementation of the response sent from server to client

```
public class TestRequest {
    String requestId;
    List<SoapTestData> requestData = new ArrayList<>();

    public String getRequestId() {
        return requestId;
    }

    public void setRequestId(String requestId) {
        this.requestId = requestId;
    }

    public List<SoapTestData> getRequestData() {
        return requestData;
    }

    public void setRequestData(List<SoapTestData> requestData) {
        this.requestData = requestData;
    }
}
```

Example 4.3. Java class implementation of the request sent from client to server

4.2 Test results and analyzes

As the background for both technologies were studied, the RESTful approach was expected to have advantage. The REST and JSON has been claimed to be more lightweight and more efficient message carrier than SOAP. For example, study made by Hatem Hamad, Motaz Saad, and Ramzi Abed show the performance of the REST to be better in terms of response time and

message size. Similarly P.A. Castillo has studied high-level application and resulted the REST to be clearly faster compared to SOAP solution. However, the implementation of the platform and parser choices makes it possible for differences in performance. [10, 11, 33]

The fetch request of data showed immediately difference between the SOAP and JSON. The results of the fetch request test set with small data packet is shown in table 4.1 and time comparison illustrated in figure 4.3. The results show the expected results terms of both packet size and respond time. Even though the efforts of trying to minimize measurement errors, the small packet size results clearly show random error appearing in results.

Table 4.1 Results of the fetch data –test with small payload size.

	SOAP fetch request			**JSON fetch request**		
Amount of objects	Data sent (bytes)	Time (ms)	Received data(bytes)	Data sent (bytes)	Time (ms)	Received data(bytes)
100	296	1,951	14379	0	0,892	10640
150	296	1,795	21429	0	0,907	15940
200	296	2,149	28479	0	0,987	21240
250	296	2,21	35529	0	1,1	26540
300	296	2,362	42579	0	0,994	31840
350	296	2,169	49629	0	1,074	37140
400	296	2,687	56679	0	1,359	42440
450	296	2,586	63729	0	1,157	47740
500	296	2,154	70779	0	1,29	53040

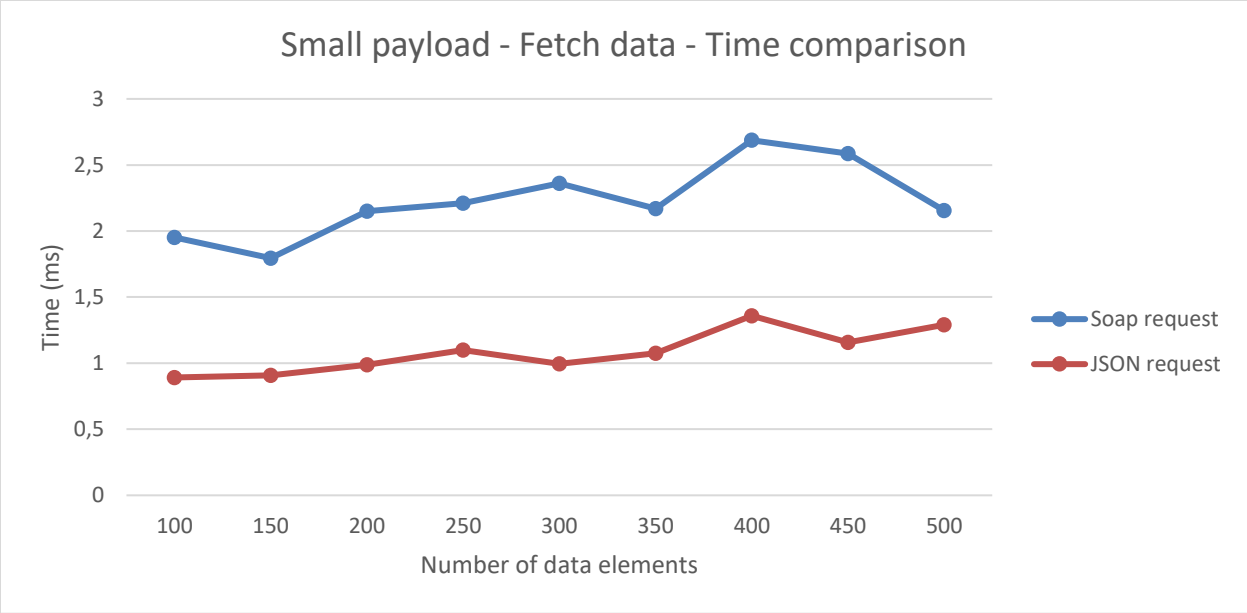

Figure 4.3 Time comparison of fetch data –test.

After the fetching tests, sending of data objects was run. The results are shown in table 4.2 and figure 4.4. Again, the results show the RESTful solution to handle the request faster, even though the difference is lesser as when fetching the data. Also the numbers show a little randomness being caused most likely by the test platform.

The small data tests show the major difference in message sizes, which was expected when moving from SOAP XML to JSON. The JSON reduces the message size by 25%. It was noticed that the

Table 4.2 Results of set data –test with small payload size.

	Soap set request			**JSON set request**		
Amount of objects	Data sent (bytes)	Time (ms)	Received data(bytes)	Data sent (bytes)	Time (ms)	Received data(bytes)
100	14161	2,100	310	10942	2,056	23
150	21111	2,353	310	16392	1,983	23
200	28061	2,736	310	21842	2,052	23
250	35011	3,022	310	27292	2,506	23
300	41961	2,635	310	32742	2,522	23
350	48911	2,567	310	38192	2,548	23
400	55861	2,857	310	43642	2,402	23
450	62811	2,637	310	49092	2,392	23
500	69761	3,042	310	54542	2,322	23

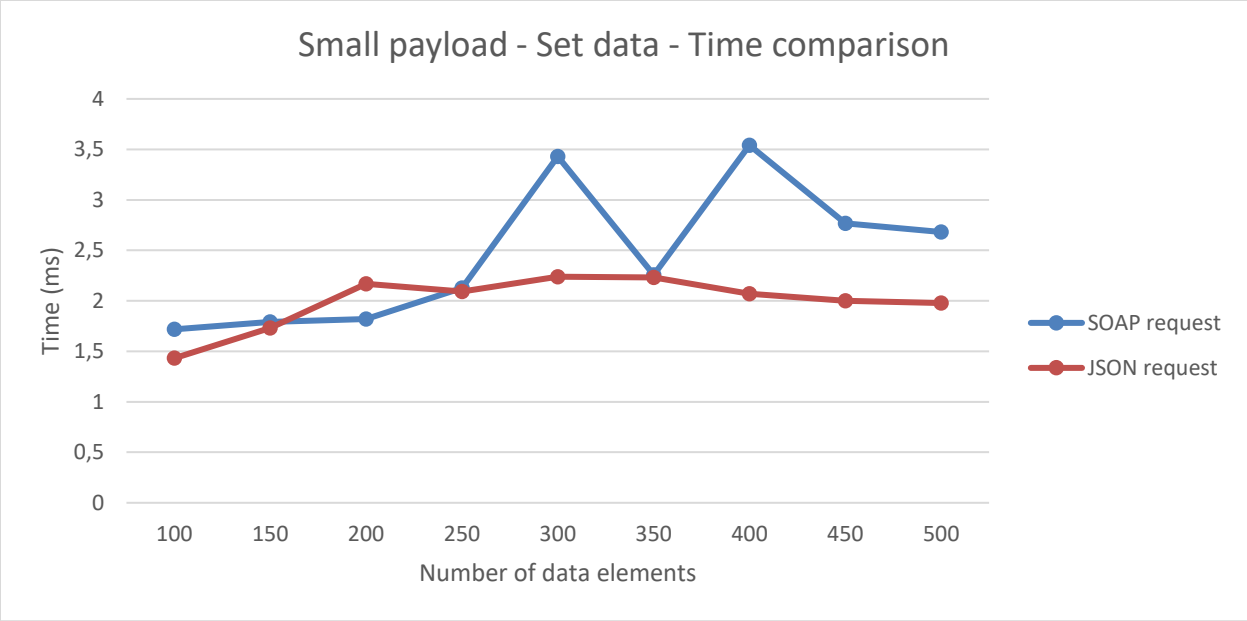

Figure 4.4 Time comparison of set data –test.

After small payload tests, the same tests were run with the larger data size. The presentation of the test results are shown in table 4.3 and figure 4.5.

Table 4.3 Results of fetch data –test with large payload size.

	SOAP fetch request			**JSON fetch request**		
Amount of objects	Data sent (bytes)	Time (ms)	Received data(bytes)	Data sent (bytes)	Time (ms)	Received data(bytes)
1000	297	4,853	141279	0	2,036	106040
3000	297	5,726	423279	0	3,884	318040
5000	297	8,979	705279	0	5,937	530040
7000	297	10,986	987279	0	7,773	742040
9000	297	11,977	1269279	0	9,906	954040
11000	298	13,635	1551279	0	10,968	1166040
13000	298	15,884	1833279	0	13,835	1378040
15000	298	17,955	2115279	0	15,768	1590040
17000	298	21,604	2397279	0	18,183	1802040
19000	298	21,438	2679279	0	21,237	2014040

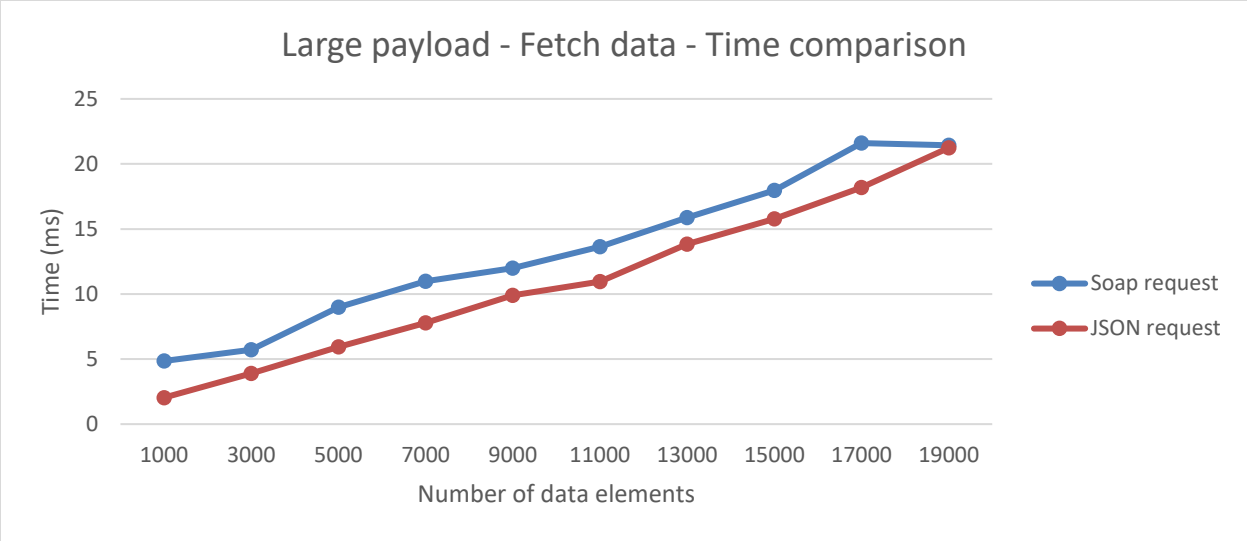

Figure 4.5 Time comparison of fetch data –test with large payload.

Larger data amount when fetching data provides very similar data as when processing small files. The JSON is clearly faster and provides smaller data packet sizes.

Table 4.4 Results of set data –test with large payload size.

	SOAP set request			**JSON set request**		
Amount of objects	Data sent (bytes)	Time (ms)	Received data(bytes)	Data sent (bytes)	Time (ms)	Received data(bytes)
1000	139261	6,295	310	106038	21,667	23
3000	417261	8,757	310	318038	23,007	23
5000	695261	12,632	310	530038	186,266	23
7000	973261	17,112	310	742038	188,288	23
9000	1251261	21,415	310	954038	190,615	23
11000	1529261	26,040	310	1166038	191,528	23
13000	1807261	29,804	310	1378038	193,463	23
15000	2085261	34,414	310	1590038	195,658	23
17000	2363261	39,889	310	1802038	197,859	23
19000	2641261	43,392	310	2014038	199,946	23

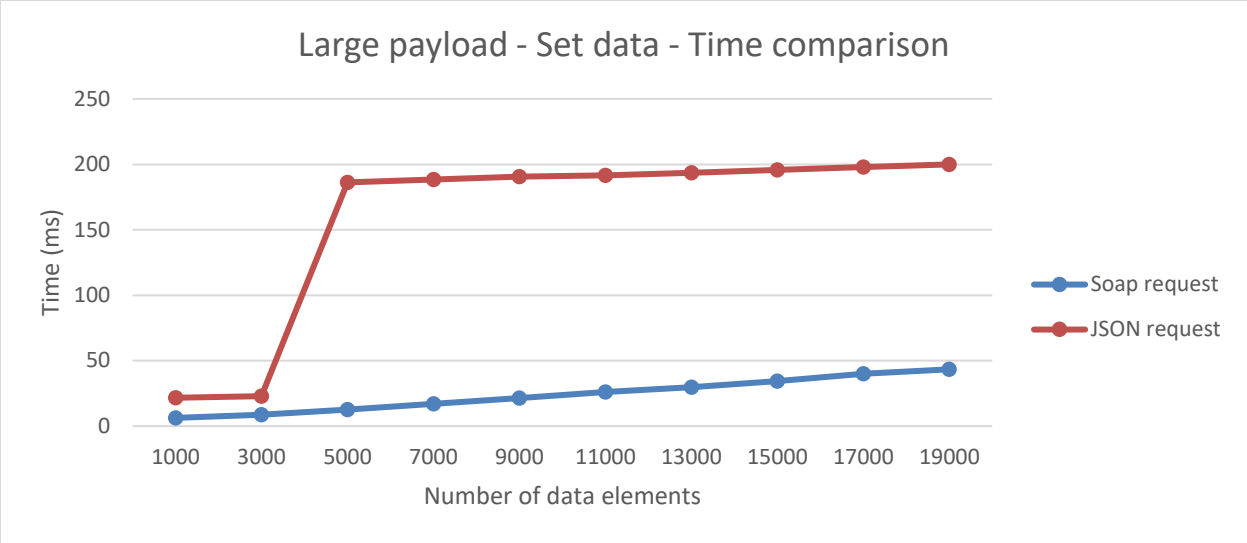

Figure 4.6 Time comparison of set data –test with large payload.

The final set data test results are shown in table 4.4 and figure 4.6, which reveals issues with JSON requests. The JSON requests take step up in processing time, while the number of payload bytes stay the below the corresponding SOAP message. As the results are not in line with other tests, some additional analyzing is required.

To find out the reason for the set data results, additional research is required. In test results the packet handling rises at 5000 data objects, where the size of the packet is almost 700kb. The options for the behavior was researched and tests reviewed for errors, but no clear reason was found. As the packet transfer delay is eliminated, the only option for the source of the delay comes from parser. The parser was studied using other studies of GSON parser and possible reason for delay was found. Studies comparing JSON parsers report GSON to have troubles when packet size rises, making it possible to be issue in the test. [12]

Overall, the test results show the REST implementation to have an advantage in terms of bandwidth. In all tests, the RESTful approach using JSON reduced the payload size by 25% from the SOAP equivalent. In addition, in fetch request the body message overhead is completely eliminated, as the request parameters are sent in the URL.

In terms of processing time, which can be read as performance as well, the test results are showing an advantage to REST, but the error chance is high. When examining results individually, the small payload tests bring up the processing speed to JSON parsing, whereas the large payload numbers are more difficult to interpret the advantage. These results are heavily affected by the parser chosen for the implementation, so additional research with other parsers should be done.

To confirm and compare the results, additional research was done against studies with similar testing. Similarly to this thesis, the studies prove the RESTful implementations to have better performance through better processing time and reduced message size and bandwidth usage. The REST has been confirmed to achieve better and recommended to even more complicated systems. [10, 13, 19, 20, 25, 30, 36]

5. Discussion

In this chapter apply the previous knowledge of REST and SOAP to discuss about their strengths and weaknesses while trying to keep in mind the payment solution during the comparison. We try to do the comparison in both point of views: common web service designing and implementation, and Seitatech payment solution. The comparison of REST and SOAP in definition level is difficult, as the technologies are different in definition: SOAP is a protocol, whereas REST is architectural style or a set of architectural rules.

5.1 Differences of REST and SOAP

The differences can be divided as conceptual and feature differences. Conceptual differences affect to early phase design decisions, where the technological decisions are not yet taken into account. For the technology decisions which are often considered later in software development, we do a feature difference comparison.

When going through the aspect listed in table 5.1, we first take a look to the technologies as a concept. As mentioned before, SOAP and REST are not straightforward comparable, since they are very different in terms of software architecture and design. As the REST relies on the definition of constraints, it is more of a guideline for designing web services, thus leaving much space for decisions for low level designing. SOAP on the contrary defines the structure and the protocol of the messaging very accurately, completely leaving out the rest of the web service design.

The ideology of operation is different between the technologies. REST deploys the service using the concept of resource, whereas SOAP is using operations to access the data required. Both styles have advantages. Resource handling of REST allows easy tracking of used data, while SOAP offers reliable processing of certain task.

Performance was the aspect already proven in this work and similar studies. REST performs better and reduces unnecessary network traffic, especially when handling great number of small messages. Scalability is highly performance related aspect, where REST is also taking lead in as the limited layering allows scalable service designing.

One practical aspect is the complexity of the technology. Developers often give value to technologies which are easier to learn and add to their service. As the both technologies are studied in chapters 2 and 3, it can be stated that SOAP is more complex and has strict rules to follow, whereas REST definition is teaching the concepts and leaving the implementation to user. On the other hand, when the studying is done and technology is well known, strict rules are easy to follow.

For the conceptual part, the overview of the comparison stands as heavier SOAP versus lighter REST, which can be said to be true in every aspect we have studied here. The REST allows freedom of choice and improves performance, while also enables the scalability of the system.

Table 5.1. Conceptual difference comparison between SOAP and REST.

Aspect	REST	SOAP
Definition	Architectural style	Messaging protocol
Logic	Resources	Operations
Scalability	Good	Limited
Standards	Loose standards only	Strict standards defined
Message readability	Readable JSON or XML	Not easily readable
Communication	Point to point styled	Distributed systems
Performance	Faster	Slower
Complexity	Easy	Difficult
Documentation	Easier to understand	More complex

The summary of valuable feature comparison can be seen in table 5.2. As mentioned previously, the features listed here are mainly limitations for technology decisions when designing the software.

The greatest differences are the way to transport and package the messages. Even though the REST is not strictly forced to use HTTP, the SOAP can be considered to have more options for message sending method. The flexibility of SOAP is increased further as the transport type may change during the transport of one message. As for the message format, REST brings flexibility by not setting any restrictions. The base of the SOAP is in the SOAP-XML format, which while providing detailed message format, it burdens the message by making it heavier and more difficult for human to read. Finally, REST is less limiting and allows more technologies to be used.

More functional feature are caching and failure handling. The REST enables use of message caching, which can save the amount of message sent to the network. This is often valuable feature, especially when similar messages without dynamic content is often requested. Failure handing has two different solutions: client or server -side handling. Client side handling is here more beneficial in terms of performance, as it does not stress the server machine. The downside with this is the moving of complexity to client software, which may be an issue if client is developed by third party developer. In addition, as the error situations produce low size error messages, SOAP XML overhead is again an extra burden in network traffic.

When speaking of security, both technologies have their ways, even though SOAP is better prepared with heavier tools. SOAP uses the extensibility to allow security module: Web Service Security (WS-Security), which is standardized and provides message level encryption for message. REST handles the security using HTTPS (TLS/SSL), which applies security to transfer level meaning the encryption to apply when messages is transferred between the server and client. In this manner, the HTTPS can be considered less secured compared to WS-Security. For compensation, the HTTPS solution is easier to setup and use. Note that the SSL is also available for SOAP as well. [14, 15]

Table 5.2 Technology comparison between REST and SOAP.

Feature	**REST**	**SOAP**
Transport	HTTP	Various choices
Method	GET, POST, PUT, DELETE	WSDL
Security	HTTPS	HTTPS, WS-Security
Message format	JSON, XML, plain text	XML
Caching	Able to cache	Not able to cache
Failure handling	Client side	Server side

While studying these technologies during this work, we noticed that the concepts of REST are easier to understand than the standards and definitions of SOAP. The REST concepts are more practical and abstracted, which allowed me to forget the implementation problems when studying for architecture. Even when the implementation was required for the tests, using JSON was easier when constructing the messages.

Finally, as the differences between the technologies are resolved, there are differences in the usage to account for. The SOAP has been used for a long time, which has developed more useful tools to help the development of the web service. Even though the cap is getting smaller, the lack of tools might come an issue and slow down the development significantly.

Overall, the REST stands as modern, flexible and simplified architecture for web services. SOAP represents the complex, but more familiar protocol with variety of tools available. The differences give the picture of more modern REST to be superior in most of the cases, unless a particular functionality is required from SOAP. When choosing SOAP over REST, drawbacks of technology limitations must be taken in the account.

For Seitatech point of view, the REST could be used for simple services, but as the target was high security and reliability requiring service, it might be necessary to stay in SOAP format. If the security can be solved in other ways, scalability of REST would offer great advantage for future.

5.2 Choosing between REST and SOAP

To make the decision between SOAP and REST, following questions can be used for choosing between the SOAP and REST:

- Is flexibility required for API design?
- Is the service required to grow in future?
- Client type: Is lightweight required?
- What level of security is required?

After these questions are answered, following example cases can be recommended:

- Public API: REST is strong and easy to use, while HTTP is web browser friendly. Also the statelessness help the development as user or server developer does not need to worry about the control.
- Complicated internal systems: Usually SOAP is recommended for more complicated system, but both approaches are available. The formal contracts of SOAP simplify the system and possibility to change the transfer method enables creation of internal systems.
- Reliability and security required systems: If the security of HTTPS is not enough, SOAP is the correct choice. SOAP also has tools for increasing the reliability through pathing logic, where REST relies on client side to retry the request.
- Legacy service: If service is already using SOAP as a part of the message handling, the new services may be forced to use SOAP.

To summarize, REST is easier to develop and use than SOAP and should be the default option when developing web service. The SOAP should be considered when a particular feature of SOAP is needed and the complexity is not an issue.

6. Conclusions

In this thesis, we studied SOAP and REST by their definitions, which were used to make an estimation of the differences between the technologies. REST appeared as a guideline to create a web service without creating much of restriction for technology choices. On the other hand, SOAP definition provided an accurate instruction of message format and ways how the message should be transferred and processed. Both technologies proved promise for web service message exchange.

After the definition studies, the technologies were used in practice by performing tests, where the qualities of REST and SOAP was compared. The tests run on a Seitatech provided platform, which added additional value by providing information for payment application web service. The test application was created and the message processing performance and bandwidth usage was chosen as interesting metrics for the tests. The REST was decided to use HTTP as message transfer method and JSON as message format. Similarly SOAP holds HTTP for transferring, while the SOAP XML was used as message format.

The results of the testing was clear and verified REST to perform better in terms of both bandwidth usage and message handling. The tests show the benefits of REST in forms of shorter messages compared to XML, causing less bandwidth usage. While the overall results proved REST to process messages faster, issues were discovered as size of the test message increased. Estimates for this behavior was done and the JSON parser used in Seitatech platform appears to be the reason.

The comparison provided results of characteristic differences between SOAP and REST. SOAP is more traditional and heavy, while REST is modern and lightweight way to develop web service message delivery. Advantages of REST consists of scalability, freedom of choice for message format and ease of deployment, whereas it suffers from lack of tools and security options. SOAP suffers from complexity and burdening style of message handling, but brings reliability and extensibility. Overall, REST outperforms SOAP in most of the critical areas. The recommendation is to use REST unless extra security or reliability is required.

References

[1] Fielding, R. T. REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

[2] Richardson, L., and Ruby, S. Restful web services, 1.st ed. O'Reilly,2007.

[3] T. Berners-Lee, R. Fielding, L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. Network Working Group. http://www.hjp.at/doc/rfc/rfc3986.html. Cited 12.9.2017.

[4] Mark Massé. REST API Design Rulebook, 1.st ed. O'Reilly,2011.

[5] James Snell. Programming web services with SOAP. 1.st ed. O'Reilly, 2001.

[6] Mitra, N., and Lafon, Y. SOAP version 1.2 part 0: Primer (second edition). W3C Recommendation, W3C, 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427. Cited 12.9.2017

[7] XML 1.0 Specification. W3C Recommendation, W3C, 2008. https://www.w3.org/TR/REC-xml/. Cited 12.9.2017.

[8] SOAP Version 1.2 Part 2: Adjuncts (Second Edition). W3C Recommendation, W3C, 2007. https://www.w3.org/TR/2007/REC-soap12-part2-20070427/. Cited 12.9.2017

[9] Network Working Group. HTML specification. https://tools.ietf.org/html/rfc2616. Cited 12.9.2017.

[10] Hatem Hamad, Motaz Saad, and Ramzi Abed. Performance Evaluation of RESTful Web Services for Mobile Devices. International Arab Journal of e-Technology, Vol. 1, No. 3, January 2010.

[11] P.A. Castillo, J.L. Bernier, M.G. Arenas, J.J. Merelo, P. Garcia-Sanchez. SOAP vs REST: Comparing a master-slave GA implementation. arXiv:1105.4978 [cs.NE]. 2011.

[12] Kazuaki Maeda. Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats. IEEE, DICTAP, Second International Conference. 2012.

[13] G. Mulligan, D. Gracanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. Winter Simulation Conference. 2009.

[14] Web Services Security: SOAP Message Security Version 1.1.1. Oasis Standard. 18 May 2012. http://docs.oasis-open.org/wss-m/wss/v1.1.1/os/wss-SOAPMessageSecurity-v1.1.1-os.html. Cited 12.9.2017.

[15] E. Rescorla. HTTP Over TLS. Network Working Group. May 2000. https://tools.ietf.org/html/rfc2818.

[16] Mitra, N., and Lafon, Y. SOAP version 1.2 part 0: Primer (second edition). W3C Recommendation, W3C, 2007. https://www.w3.org/TR/2007/REC-soap12-part0-20070427/. Cited 12.9.2017.

[17] JSON specification. Internet Engineering Task Force. March 2014. https://tools.ietf.org/html/rfc7159. Cited 12.9.2017.

[18] Bruno Costa, Paulo F. Pires, Flávia C. Delicato. Evaluating a Representational State Transfer (REST) Architecture. In IEEE/IFIP Conference on Software Architecture, 2014.

[19] Snehal Mumbaikar, Puja Padiya. Web Services Based On SOAP and REST Principles. International Journal of Scientific and Research Publications, May 2013.

[20] Feda AlShahwan, Klaus Moessner. Providing SOAP Web Services and RESTful Web Services from Mobile Hosts. Centre for Communications Systems Research, 2010.

[21] Network Working Group. Hypertext Transfer Protocol 1.1. http://www.ietf.org/rfc/rfc2616.txt. Cited: 12.11.2017.

[22] Xiwei Xu, Liming Zhu, Yan Liu, Mark Staples. Resource-Oriented Architecture for Business Processes. 15th Asia-Pacific Software Engineering Conference. 2018.

[23] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing Web services choreography standards - the case of REST vs. SOAP. Decision Support Systems. July 2005.

[24] Pautasso, C., Zimmermann, O., and Leymann, F. Restful webservices vs. "big"' web services: making the right architectural decision. In Proceeding of the 17th international conference on World Wide Web. 2008.

[25] Philip Markey, Gary Clynch. A performance analysis of WS-* (SOAP) and RESTful Web Services for Implementing Service and Resource Orientated Architectures. The 12th Information Technology and Telecommunications (IT&T) Conference. 2013.

[26] Mark Massè. REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly. 2012.

[27] GSON documentation. https://sites.google.com/site/gson/Home. Cited 12.11.2017.

[28] Subbu Allamaraju. RESTful Web Services Cookbook. O'Really. March, 2010.

[29] Robert Daigneu, Ian Robinson. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley Professional. October, 2011.

[30] Fatna Belqasmi, Jagdeep Singh, Suhib Younis Bani Melhem, Roch H. Glitho. SOAP-Based vs. RESTful Web Services A Case Study for Multimedia Conferencing. IEEE Computer Society. Vol 16, issue 4. May, 2012.

[31] Leonard Richardson, Mike Amundsen, Sam Ruby. RESTful Web APIs: Services for a Changing World. O'Reilly. 2013.

[32] Xinyang Feng, Jianjing Shen, Ying Fan. REST: An Alternative to RPC for Web Services Architecture. First International Conference on Future Information Networks. 2009.

[33] Snehal Mumbaikar, Puja Padiya. Web Services Based On SOAP and REST Principles. International Journal of Scientific and Research Publications. Vol 3, issue 5. May 2013.

[34] Brian Suda. SOAP Web Services. University of Edinburgh. 2003.

[35] Ethan Cerami. Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. O'Reilly. 2002.

[36] Pavan Kumar Potti, Sanjay Ahuja, Karthikeyan Umapathy, Zornitza Prodanoff. Comparing Performance of Web ServiceInteraction Styles: SOAP vs. REST. Proceedings of the Conference on Information Systems Applied Research. 2012.

[37] Pavan Kumar Potti. On the Design of Web Services: SOAP vs. REST. UNF Theses and Dissertations. 2011.

[38] Fatna Belqasmi, Roch Glitho, Chunyan Fu. RESTful Web Services for Service Provisioning in Next-Generation Networks: A Survey. IEEE Communications Magazine. December, 2011.

[39] Haibo Zhao, Prashant Doshi. Towards Automated RESTful Web Service Composition. IEEE International Conference on Web Services. 2009.

[40] Jian Meng, Shujun Mei, Zhao Yan. RESTful Web Services: a Solution for Distributed Data Integration. International Conference on Computational Intelligence and Software Engineering. December, 2009.

[41] S. Vinoski. RESTful Web Services Development Checklist. IEEE Internet Computing. November, 2008.

[42] Erik Wilde, Cesaro Pautasso. REST: From Research to Practice. Springer Science+Business Media. 2011.

[43] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, Sanjiva Weerawarana. Unraveling the Web Services Web An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing. 2002.

[44] Li Li, Wu Chou. Design patterns for restful communication web services. IEEE International Conference on Web Services. 2010.